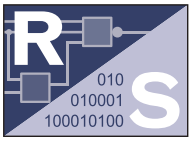
Bericht von Rechnersystem II SoC Versuch

Seminararbeit von Sharif Azem 2316873, Jingang Zhang 2705413
Tag der Einreichung: 5. Juni 2023

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT



Fachbereich Elektrotechnik und Informationstechnik
Institut für Datentechnik
FG Rechnersysteme

Inhaltsverzeichnis

1 Einführung

In diesem Bericht wird die Anwendung und Implementierung von Inter-Integrated Circuit (I2C) und Serial Peripherial Interface (SPI) näher betrachtet. Der I2C-Treiber wird zur Kommunikation mit einem Abstandssensor verwendet, Während der SPI-Treiber zur Ansteuerung eines Displays eingesetzt wird. Darüber hinaus wird ein Medianfilter implementiert, um die rauschbehaftete Messungen zu verbessern. Außerdem kommt bei der I2C Kommunikation Interupt Service Routine (ISR) zum Einsatz.

I2C und SPI sind serielle Kommunikationsprotokolle, die in Informationstechnischen Systemen oft verwendet werden, um Verbindung und Kommunikationsaustauch zwischen zwei oder mehreren Komponenten zu ermöglichen. Dabei übernimmt eine Komponente die Rolle des Masters und die anderen Komponeneten die Rolle des Slaves. Diese Kommunikationsprotokolle ermöglichen einen zuverlässigen Datenaustausch zwischen den verschiedenen Komponenten.

Im vorliegenden Versuch liegt der Fokus der Implementierung eines I2C Treiber zur Kommunikation mit einem Abstandssensor (I2C Slave). Der Sensor erfasst Messungen und stellt sie dem Master zur Verfügung. Diese Messungen sollen auf einem Display (SPI Slave) angezeigt werden. In diesem Versuch wird nur die Master Seite implementiert.

Die Abstandsmessungen enhalten Außreiser. Um denoch sinvolle Ergebnisse anzuzeigen, wird ein Medianfilter implementiert. Der Medianfilter ist ein digitales Signalverarbeitungsverfahren das unerwünschte Eigenschaften des Rauschens reduziert. Dadurch kann die Qualität der Abstandswerte verbessert werden.

Ziel diese Versuchs besteht darin, die I2C und SPI Kommunikation auf einem System zu integrieren, sodass eine erfasste Messung auf dem Display angezeigt wird. Dafür teilen wir das Program in drei verschiedene Teilsysteme, nämmlich das Abstandsmessung-Programteil mit der I2C Funktionalität, das Display-Programteil mit der SPI Dunktionalität und der Filter-Programteil. wir implementieren beide Kommunikationsprotokolle zunächst unabhängig von einander und danach integrieren wir sie zusammen in einem Program.

Dieser Bericht ist wie folgt strukturiert. In Kapitel 2 diskutieren wir die Implementierung des SPI Masters zur Kommunikation mit dem Display. In Kapitel 3 widmen wir uns der Realisierung des I2C Masters, das zur Kommunikation mit dem Abstandssenor benötigt wird. In Kapitel 4 integrieren wir beide Teilsysteme. Kapitel 5 fasst den Versuch zusammen.

2 Display

Die SPI Schnittstelle besteht aus einer Slave Peripherie (OLED Display NHD-2.8-25664UCB2) und Mastermodul, der zur Ansteuerung des Slaves verwendet wird. Der Master besitzt 2 Register, die durch Bitmanipulationen in verschiedene Zustände gesetzt werden können. Ein weiteres Register wird zum Schreiben der Information, die auf dem Display angezeigt werden soll.

Um die Information zu übertragen, muss die SPI Peripherie eingeschaltet werden. Dafür wird zunächst das Controlregister zurückgesetzt, in dem die Zahl null reingeschrieben wird und dann das erste Bit auf 1 gesetzt wird. Danach setzt man den Anzahl an bits, die Übertragen werden sollen. In unserem Fall sind es 9 bits, wobei das Most Significant Bit (MSB) verwendet wird, um festzustellen, ob es sich um ein Befehl oder Information handelt, die auf dem Display angezeigt werden soll. In dem Kontrollregister setzten wir dann die bits 8 bis 12 dementsprechend. In bits 13 bis 15 setzt man die Frequenz, mit der der Master arbeiten soll, um mit dem Prozessorkern Synchronisiert zu sein. Der Prozessorkern arbeitet mit $60MHz$, während das SPI Clock mit bis zu $10MHz$ arbeiten kann. Deshalb setzen wir in das bit 14 auf 1. Die Formell zur Berechnung von dem Clock Divider seht in dem Datenblatt von Spartanmc.

$$\text{SCLK} = \frac{\text{peripheral\_clock}}{4 \times \text{divider}} \tag{2.1}$$

Das oben beschriebene Vorgehen ist in der Funktion spi_peri_enable implementiert.

```
void spi_peri_enable(spi_master_regs_t* spi_master)
{
    spi_master->spi.spi_control &= 0;
    spi_master->spi.spi_control |= SPI_MASTER_CTRL_EN;
    spi_master->spi.spi_control &= ~SPI_MASTER_CTRL_BITCNT;
    spi_master->spi.spi_control |= (9<<8);  // set the number of bits in one frame
    to 9
    spi_master->spi.spi_control |= (1 << 13); // set the clock devider
}
```

SPI Enable Funktion

Der SPI Master kann mit mehreren Slaves Kommunizieren. Da derselbe Datenbus verwendet wird, muss bei jeder Kommunikation festgelegt werden, um welchen Slave es sich handelt, um diesen die Information zu schicken. Bits $4-7$ vom Controlregister sind dafür da, um die richtige Slaveadresse zu setzen. Da der Master mit nur einem Slave kommuniziert und die SPI-Schnittstelle im sogenannten 3-wire Modus betrieben wird, setzen wir Bit 5 auf 1. Die Implementierung wird in der Funktion spi_peri_select gezeigt.

```c
void spi_peri_select(spi_master_regs_t* spi_master, const unsigned int slave)
{
    /* activate the slave */
    spi_master->spi.spi_control |= (slave << 4);
}
```

SPI Select Funktion

Nachdem der Master das Controlregister richtig eingestellt hat, kann die Information in das Datenregister geschrieben werden. Davor wird gewartet, bis der aktuelle Kommunikationsvorgang beendet. Das wird geprüft, indem man das Statusregister in einer While-Schleife abfragt. Nachdem das Statusregister das Ende der Kommunikation zeigt, schreibt der Master die Information in das Datenregister. Dieses 9 bit lang Informationsframe wird dann vom Slave bearbeitet. Dabei kann es sich um einen Befehl oder eine Nachricht handeln.

```c
void spi_peri_write(spi_master_regs_t* spi_master, unsigned int dt)
{
    /* data written into spi_data_out will be sent */
    while(spi_master->spi.spi_status & SPI_MASTER_STAT_FILL);//polling
    spi_master->spi.spi_data_out = dt;
}
```

SPI Write Funktion

In der Firmwareebene sind zahlreiche Funktionen implementiert um verschiedene Zeichen auf dem Display anzuzeigen. Diese Funktionen verwenden die oled_Command_25664 um Befehle zu senden und die Funktion oled_Data_25664 um anzuzeigebde Information zu senden. Diese Funktionen verwenden die spi_peri_write Funktion, nachdem das erste Bit richtig gesetzt wurde (0 bei Befehl und 1 bei Anzeigeinformation). Wenn es sich um einen Befehl handelt, dann führt der Slave diesen Befehl aus. Wenn es sich um Information handelt, dann wird diese auf dem Display gezeigt.

```c
void oled_Command_25664(unsigned char Data)
{
    unsigned short my_data = Data & 0x0FF;
    spi_peri_write(&SPI_MASTER, my_data);
}
```

Oled Command Funktion

```c
void oled_Data_25664(unsigned char Data)
{
    unsigned short my_data = Data | 0x100;
    spi_peri_write(&SPI_MASTER, my_data);
}
```

Oled Data Funktion

Um die Kommunikation mit dem Slave zu beenden, müssen die Bits $4 - 7$ wieder auf Null gesetzt werden. Die Implementierung dazu liegt in der Funktion spi_peri_deselect.

```
void spi_peri_deselect(spi_master_regs_t* spi_master)
{
    /* deactivate the slave */
    while(spi_master->spi.spi_status & SPI_MASTER_STAT_FILL);//polling
    spi_master->spi.spi_control &= ~SPI_MASTER_CTRL_SLAVE;
}
```

SPI Deselect Funktion

Die unten stehende Main Funktion zeigt, wie die SPI Schnittstelle verwendet wurde, um "Hello World" auf dem Display anzuzeigen. Zunächst definieren wir ein Array vom Typ char, der die Buchstaben enthält. Danach wird die SPI Schnittstelle eingeschaltet und ein Slave wird gewählt und das Display initialisiert. Schließlich wird die Information an das Display geschickt und die Kommunikation beendet.

```
void main()
{
    char *data = "Hello world";
    while(1)
    {
        spi_peri_enable(&SPI_MASTER);
        spi_peri_select(&SPI_MASTER, 0x01);
        OLED_Init_25664();
        Show_String_25664(1, data, 10, 10);
        spi_peri_deselect(&SPI_MASTER);
    }
}
```

SPI Main Funktion

3 Abstandsmessung

Für die Kommunikation mit dem Abstandssensor benötigen wir den I2C-Treiber. In diesem Abschnitt diskutieren wir die Implementierung des I2C-Masters. Die Hauptaufgabe besteht darin, eine bidirektionale Datenübertragung über das I2C-Modul zu implementieren.

Um die I2C-Kommunikation zu ermöglichen, muss der Kontrollregister richtig eingestellt werden. Die Funktion i2c_peri_enable zeigt, wie der Kontrollregister eingestellt wird. Mit der konstante I2C_CTRL_EN wird die Kommuniktation freigegeben, mit der I2C_CTRL_INTR_EN Konstante werden Interrupts ermöglicht und mit PRESCALER_FACTOR wird der Prescaler gesetzt und somit wird der I2C-Bus richtig getaktet. Die Taktfrequenz für die SCL-Leitung des I2C-Busses ist $40KHz$, während der Prozessorkern (SpartanMC) mit einer Frequenz von $60MHz$ getaktet wird. Der Prescaler-Faktor(299) des I2C-Control-Registers kann mit der folgenden Formel bestimmt werden:

$$\text{Prescaler-Faktor} = \left(\frac{\text{peripheral\_clock}}{5 \times \text{desired\_SCL}}\right) - 1 \tag{3.1}$$

```
1 #define PRESCALER_FACTOR 299
```

Definition des Prescaler-Faktors

```
1 void i2c_peri_enable(i2c_master_regs_t* i2c_master)
2 {
3     i2c_master->ctrl = 0;
4     i2c_master->ctrl = I2C_CTRL_EN | PRESCALER_FACTOR | I2C_CTRL_INTR_EN;
5 }
```

I2C Enable Funktion

Da mehrere Slaves an den I2C-Bus angeschlossen werden können, ist es wichtig, die Adresse vom Slave richtig einzustellen. In diesem Versuch haben wir nur einen einzelnen Slave, daher wählen wir die Adresse des SRF02 $(0xE0)$ als die Slave-Adresse. Diese Adresse wird dann in das Datenregister geschrieben. Wenn es sich um einen Lesevorgang handelt, müssen wir 1 der Addresse hinzufügen. Beim Schreiben bleibt die Adresse unverändert. Um eine Messung zu starten, müssen wir noch das Befehlsregister vom Sensor $(0x00)$ angeben und den richtigen Befehl dazu $(0x51)$. Die Funktion i2c_peri_write zeigt die Implementierung dazu. Nachdem wir das Kontrollregistergeschrieben haben, warten wir in der while Schleife bis die Übertragung fertig ist.

```
1 #define I2C_SLAVE_ADDRESS 0xE0
```

Definition der Slave-Adresse

Acht Daten-Register speichern die zu sendenden oder empfangenen Daten. Mit dem Command-Register können die Anzahl der Datenbytes und die Richtung der Datenübertragung (lesen oder schreiben) konfiguriert werden. Es ist wichtig zu beachten, dass das erste Byte immer die Slave-Adresse ist und geschrieben wird.

```
void i2c_peri_write(i2c_master_regs_t* i2c_master)
{
    i2c_master->cmd = 0; //clear all command register bits zero
    i2c_master->data[0] =  I2C_SLAVE_ADDRESS; //0xE0
    i2c_master->data[1] = 0x00; // Befehlregister
    i2c_master->data[2] = 0x51;
    // send write request to the slave
    i2c_master->cmd = I2C_CMD_STA | (3 << 3) | I2C_CMD_STO;
    // check if transfer is finished, polling
    while(i2c_master->stat & I2C_STA_TIP);
}
```

I2C Write Funktion

Wenn wir Bytes vom Slave lesen wollen, muss zuerst die Registernummer vom Sensor geschickt werden. Das ist in der Funktion i2c_peri_set_sense_register implementiert.

```
void i2c_peri_set_sense_register(i2c_master_regs_t* i2c_master, unsigned int
   reg_num)
{
    i2c_master->data[0] = I2C_SLAVE_ADDRESS;
    i2c_master->data[1] = reg_num;
    i2c_master->cmd = I2C_CMD_STA | (2 << 3) | I2C_CMD_STO;
    // check if transfer is finished, polling
    while(i2c_master->stat & I2C_STA_TIP);
}
```

Select Highbyte-Register

Mit der Funktion i2c_peri_read_start starten wir den Lesevorgang. Am Ende dieses Vorgangs stehen 2 in dem Datenregister, mit denen wir den gemessenen Abstand bestimmen können.

```
void i2c_peri_read_start(i2c_master_regs_t* i2c_master, unsigned int slave_add)
{
    i2c_master->cmd = 0;
    i2c_master->data[0] = slave_add + 1;
    i2c_master->cmd = I2C_CMD_RD | I2C_CMD_STA | (3 << 3) | I2C_CMD_STO;
    // check if transfer is finished, polling
    while(i2c_master->stat & I2C_STA_TIP);
}
```

Read 2 Bytes from slave

Der Abstand wir mit Formel 3.2 berechnet. Dabei ist Register2 die Information von dem Highbyte-Register und Register3 die vom Lowbyte-Register.

$$\text{Entfernungswert} = (\text{Register2} * 256) + \text{Register3} \tag{3.2}$$

```c
unsigned int i2c_peri_read_distance(i2c_master_regs_t* i2c_master)
{
    unsigned int register_2, register_3, distance=0;
    //Entfernungswert(high byte) in register_2
    register_2 = i2c_master->data[1];
    //Entfernungswert(low byte) in register_3
    register_3 = i2c_master->data[2];
    distance = (register_2*256) + register_3;
    return distance;
}
```

Read distance function

Die folgende Main Funktion zeigt wie die Kommunikation verwendet wird und anschließend am PC angezeigt wird. Dabei wird eine Sleepfunktion verwendet, um $65ms$ zu warten, bevor eine neue Messung gestartet wird. Diese Main Funktion wurde nur zum Debuggen benutzt. In der Integration wurde auf diese Funktion sowie auf die While-Schleife bei jeder Transaktion verzichtet.

```c
void main()
{
    unsigned int distance = 0;
    i2c_peri_enable(&I2C_MASTER);
    while(1)
    {
        i2c_peri_write(&I2C_MASTER);
        //Wait for 65ms, interrupt polling
        _SLEEP_MS(65);
        i2c_peri_set_sense_register(&I2C_MASTER,0x02);
        i2c_peri_read_start(&I2C_MASTER, I2C_SLAVE_ADDRESS);
        distance = i2c_peri_read_distance(&I2C_MASTER);
        printf("\nDer Abstand ist: %u cm\n", distance);
    }
}
```

Main Funktion

4 Integration Abstandsmessung und Anzeige

In diesem Kapitel geht es um die Integretion der Implementierung der Abstandsmessung und der der Anzeigefunktion. Ziel ist es, die mittels des Sensors erfassten Messungen auf der OLED Anzeige anzuzeigen. Der Vorgang kann wie folgt beschrieben werden. Es wird eine (oder mehrere) Abstandsmessung durchgeführt. Der I2C Master erhält die Messung in Form von 2 Bytes vom I2C Slave. Diese Information wird dann in einen Abstand umgerechnet und dem SPI Master zur Verfügung gestellt. Dieser schickt diese Information an die Anzeige weiter, auf der man den gemessenen Abstand ablesen kann.

Da die Messungen vom Sensor auch außreiser enthalten, müssen Signalverarbeitungsmethoden eingesetzt werden, um Sinvolle Messungen anzeigen zu können. Dafür Implementieren wir einen Filter, die Implementierung des Filters diskutieren wir auch in diesem Abschnitt.

In Kapitel 3 wird die Implementierung von den I2C Lese- und Schreibefunktion vorgestellt. Da können wir sehen, dass bei jeder Transaktion der Master mittels einer While-Schleife ständig abfragt, ob die Übertragung fertig ist. Dabei wird der Ablauf des Programs blockiert, bis die Übertragung fertig ist. Da der Master mit einer Taktfrequenz von $60MHz$ arbeitet, während der I2C-Buss mit $40kHz$ betrieben wird, können wir viel Zeit für andere Berechnungen ausnutzen, anstatt das Program zu blockieren. Hier kommen Interrupts zum Einsatz. Dafür erweitern wir die Implementierung um eine Statemachine und einen Interruptkontroller.

4.1 Integration

Die Funktionalitäten der Abstandsmessung und Anzeige zu integrieren beduetet, dass wir die Fuktionen beider Modulen in einem Program benutzen. Dieses Program können wir nun in 3 Module zerlegen. Ein Modul ist das Display-Modul. Dieses ist bereits in Kapitel 2 diskutiert und bleibt in unserem Program unverändert. Ein weiteres Modul ist das I2C-Modul. Auch dieses Modul haben wir bereits behandelt (Kapitel 3), allerdings wird die Implementierung um einen Interruptkontroller und eine Zustandsmaschine erweitert. Der dritte Teil des Programs ist der Filteralgorithmus, der die Messungen filtert und um ein verlässliches Ergebniss zu erhalten.

Um das Display-Modul anzuwenden, müssen wir die SPI-Schnittstelle initialisieren. Wir tun es in der Main-Funktion. In der Main-Funktion rufen wir auch die Funtion display_init auf, um den Satz "der Abstand ist" anzuzeigen. In der While-Schleife wird dann die Funktion

show_distance_display(distance_sens) aufgerufen, dieser wird der bereits gefilterte Distanz übergeben, und somit periodisch auf dem Display aktualisiert. Wie oft die Distannz aktualisiert wird, hängt unter anderem auch von unserer Implementierung ab. In einer Version haben wir die Distanz aktualiesiert, erst nachdem N Messungen gesammelt wurden (in der Main-Funktion die auskomentierte if Abfrage, um es zu verwenden, kommentieren wir die if Abfrage in Zustand $S3$ aus). Der Nachteil davon ist, dass die Periode, mit der die Distanz aktualisiert wird, immer mit der Anzahl an Messungen wächst. Bei der Beobachtung des Displays kam es zu erhebliche Verzögerungen bei der Aktualisierung des Wertes bei vielen Messungen ($N = 100$). In der aktuellen Version wird stattdessen die Distanz nach jeder neuen Messung aktualisiert. Dafür erzeugen wird zwei Arrays. Das Array dis_array enthält die aktuellsten Messungen. Wenn eine neue Messungen ankommt, wird diese in diesem Array an der richtigen Stelle gespeichert und ein Flag gesetzt, dass eine Messung vorhanden ist (siehe ISR Zustand $S3$). In der Main-Funktion wird der Inhalt von dis_array in das Array sort_dis_array kopiert. Danach wird sort_dis_array sortiert und gefiltert um die Distanz zu ermitteln. Schließlich wird das Flag new_val auf Null gesetzt. Somit stellen wir sicher, dass nach jeder neuen Messung der Abstandwert aktualiesiert wird. Nachteil davon ist, dass wir ein weiteres Array brauchen, und die Funktion memcpy aufrufen, die bei großen Arrays viel Zeit in Anspruch nehmen kann.

Um eine Messung zu erhalten, benutzt das Program die I2C-Schnittstelle. Das I2C-Modul ist dann für die Kommunikation mit dem Abstandssensor verantwortlich. Dieses Modul wird in der Main-Funktion initialisiert und dann wird eine Funktion aufgerufen, um eine Messung zu starten und somit ist eine Übertragung auf dem I2C Bus gestartet. Da wir einen Interruptkontroller zusammen mit dem I2C-Master verbinden, wird am Ende jeder I2C Transaktion ein Interrupt ausgelöst. Da wir nach dem ersten Aufruf die I2C Funktionen nur in der Interrupt Service Routine (ISR) aufrufen und nicht in der While-Schleife, müssen wir sicherstellen, dass die I2C-Funktionen sinnvoll in der ISR aufgerufen werden, und dass irgendwann ein Interrupt ausgelöst wird.

Der dritte Teil des Programs ist der Filter. Hier werden Ausreißer gefiltert und der Abstand ermittelt. Dieser Abstand wird dann auf dem Display angezeigt.

Die genaue Implementierung der Main-Funktion ist in dem unten Codeauschnitt zu sehen.

```
1  void main()
2  {
3      spi_peri_enable(&SPI_MASTER); // initialize the spi module
4      display_init();               // initialize the display with text
5      i2c_peri_enable(&I2C_MASTER); // initialize the i2c module
6      interrupt_enable();       // enable interrupts
7      i2c_peri_write(&I2C_MASTER);  // start a measurement
8
9      while(1) // while loop. Program jumps out of while when i2c interrupt occurs
10     {
11         /*if(meas_n >= MEAS_N) // get new distance when 10 new measurements are
       collected
12         {
13             meas_n = 0;
14             distance_sens = median_filter(dis_array, MEAS_N);
15         }*/
16         if(new_val)
```

```
17          {
18                  memcpy(sort_dis_array, dis_array, MEAS_N*sizeof(unsigned int));
19                  distance_sens = median_filter(sort_dis_array, MEAS_N);
20                  new_val = 0;
21          }
22
23          show_distance_display(distance_sens);
24      } // end of while(1)
25  } // end of main
```

Main

4.2 Filter

Die Wahl des zu verwendeten Filters hängt von dem spezifischen Anwendungsfall ab. Jeder Anwendungsfall setzt eigene Anforderungen voraus. Um diese Anforderungen festzulegen, müssen wir die statistische Eigenschaften und Frequenz des Messrauschens kennen. Beispiele für wichtige statistische Eigenschaften sind die Verteilung des Rauschens (gaußverteilt, gleichverteilt usw.), Mittelwert und Varianz und Zeitvarianz. Außerdem müssen wir die Eigenschaften des Signals kennen, das wir messen wollen. Diese Eigenschaften werden auch vom Anwendungsfall abhängen. Z.B wollen wir den Abstand zu einem sich bewegenden Objekt messen? Wenn ja, wie schnell bewegt sich das Objekt? Können auch abrupte Änderungen in der Bewegung auftreten? Können wir ein Bewegungsmodell aufstellen? Diese Informationen zum Rauschen und zum Messsignal beeinflussen die Wahl des zu verwendeten Filters.

Generell gibt es verschiedene Filtertypen, die man in Betracht ziehen kann, um Außreiser in der Distanzmessung eines Ultraschalsensors zu behandeln. Einige Beispiele sind im folgenden aufgelistet:

- Moving Average Filter: Dieser Filter berechnet den gewichteten Durschnitt der letzten N Messungen. Die Messung kann geglätet werden, indem hochfrequente Rauschkompenente gefiltert werden. Dabei kann es zu Verzögerungen kommen bei der Erkennung schneller Änderungen und wenn man die Anzahl an Messungen N und die Gewichtung ungünstig wählt, wird der Einfluss von Außreisern immernoch stark sein. Ein Beispiel für so ein Filter ist die Berechnung des Durchschnitts der letzten $N = 5$ Messungen mit dem Gewicht $\frac{1}{N}$ für alle Messungen. Da wir keinen konkreten Anwendungsfall haben und somit keine genauen Kenntnisse vom zu mesendem Signal haben und, weil die einfache Berechnung des Durchschnitts keine gute Lösung ist, wird dieser Filter in unserem Versuch nicht implementiert.
- Kalman-Filter: Der Kalman-Filter ist ein reksrusiver Filter, der den Zustand eines verrauschten Signals abschätzt. Dabei wird die Systemdynamik berücksichtigt. Dieser Filter eignet sich gut für Tracking Anwendung und für Signale mit gaußschem Rauschen. Bei nicht gaußverteiletem Rauschen ist der Filter weniger gut als andere Filter (z.B Partikelfilter). Ein Anwendungsbeispiel ist Positionierung von einer Drohne. Aufgrund des hohen Implementierungs Aufwandes und weil wir keinen konkreten Anwendungsfall haben sowie keine

genauen Daten zu dem Messrauschen besitzen, verzichten wir auf diesen Filter.

- Medianfilter: Der Medianfilter berechnet den Median aus den letzten N Messungen. Dabei muss man nur die Fensterbreite N richtig einstellen, um gute Ergebnisse zu erzielen. Dieser Filter ist effektiv bei der Entfernung von Außreisern. Aufgrund seiner einfachen Implementierung und weil wir in unserem Versuch nur Außreiser berücksichtigen, werden wir nur diesen Filter berücksichtigen.

4.2.1 Medianfilter

Im folgenden besprechen wir die Implementierung des Medianfilters. Diese ist recht einfach. Man muss zunächst die Anzahl an zu berücksichtigenden Messungen N bestimmen, diese bleibt dann zur Laufzeit des Programs unverändert. Der Ablauf läuft dann wie folgt: es werden N Messungen gesammelt, aus diesen berechnet der Filter den Median und gibt die resultierende Distanz zurück. Bei der Berechnung des Medians muss man berücksichtigen, ob N gerade oder ungerade ist, weil es für die Berechnung wichtig ist. Die Formel für die Medianberechnung sehen wir in Gleichung 4.1. Dabei ist X die Liste mit N Messungen. Die Zahl N haben wir heuristisch gewählt. Wir haben getestet, wie sich das Ergebnis für $N \in 1, 2, 10, 15, 20, 100$ ändert, in dem wir den Abstand vom Display abgelesen haben. Dabei haben wir den Sortieralgorithmus Bubble Sort verwendet und wir konnten keinen Unterschied für $N \geq 10$ festellen. Wir haben dann $N = 10$ gewählt, da wir die Liste der Messungen so klein wie möglich haben wollen, weil wir damit wenig Speicherplatz verbrauchen und die Anzeige häufiger aktualisieren.

$$\text{Median} = \begin{cases} X\left(\frac{N-1}{2}\right) & N \text{ ungerade} \\ \frac{X(\frac{N}{2}-1)+X(\frac{N}{2})}{2} & N \text{ gerade} \end{cases} \tag{4.1}$$

In Gleichung 4.1 berechnen wir den Median aus einer List X. In unserer Implementierung wird $n$te Messung in den n Eintrag dieser Liste geschrieben. Dass bedeutet, dass wir diese Liste absteigend oder aufsteigend sortieren müssen, bevor wir den Median berechnen. Hier kommt ein Sortieralgorithmus zum Einsatz. Die Wahl des Sortieralgorithmus besprechen wir in Abschnitt 4.2.2.

4.2.2 Sortieralgorithmen

Die Wahl des Sortieralgorithmus hängt von Größe der Liste und von der Anwendung ab. Mache Algorithmen sind durch ihre gute Zeiteffizienz charakterisiert, allerdings verbrauchen sie relativ viel Speicherplatz im Vergleich zu anderen Algorithmen. Andere Algorithmen verbrauchen weniger Speicherplatz aber sind weniger effizient beim Sortieren, und brauchen deshalb mehr Zeit. Wie gut ein Algorithmus ist, hängt auch davon ab, ob die Liste bereits sortiert ist und von der Größe der Liste.

Im folgenden sind zwei der gängigen Sortieralgorithmen beschrieben:

- Quicksort: Ein rekrusiver Algorithmmus, der die Einträge der Liste mit einem ausgewähltem Pivotelement vergleicht und sortiert. Die Liste wird in zwei Teillisten um das Pivot sortiert, wodurch das Pivot an die richtige Stelle gebracht wird. Dieser Vorgang wird für die Teilisten wiederholt, bis die gesamtliste sortiert ist. Um die beste Zeiteffizienz zu erzielen, muss das Pivot in jedes Mal richtig gewählt werden. Da ist die Zeitkomplexität $O(log(N))$. Wenn das Pivot jedes Mal ungünstig gewählt wird, ist die Zeiteffizienz $O(N^2)$. Die Platzeffizienz ist immer $O(Nlog(N))$. Die Daten zu Effizienz wurden [2] entnommen.
- Bubble Sort: Dieser Algorithmus vergleicht benachbarte Elemente und tauscht sie wenn nötig. Er is relativ einfach zu implementieren aber nicht effizient bei großen Listen. Best Case: Die Liste ist bereits sortiert. Zeitkomplexität: $O(N)$, Platzeffizienz: $O(1)$. Worst Case: Die Liste ist in umgekehrter Reihenfolge sortiert. Zeitkomplexität: $O(N^2)$, Platzeffizienz: $O(1)$. Die Daten zu Effizienz wurden [1] entnommen. Aufgrund der einfachen Implementierung und weil unsere Liste relative klein ist ($N = 10$), haben wir diesen Algorithmus implementiert und meistens benutzt.

In der Literatur kann man weitere Sortieralgorithmen finden. In diesem Versuch haben wir die Algorithmen Bubble Sort und Quicksort verglichen. Vor allem haben wir versucht Unterschiede festzustellen, in dem wir die Abstandswerte von der Anzeige abgelesen haben, dabei haben wir ein Objekt vor den Sensor bewegt um zu sehen, wie die Anzeige aktualisiert wird. Wir konnten keinen Unterschied feststellen. Dies lässt sich damit begründen, dass die Liste klein ist oder damit, dass die Liste einigermaßen bereits sortiert ist.

Der sortieralgorithmus wird in unserem Program in der Funktion des Filters aufgerufen und nur dort verwendet.

4.3 Erweiterung um Interrupts

Der Ablauf von dem Hauptprogram ohne Interrupts kann (vereinfacht) wie folgt beschrieben werden:

1. N Messungen sammeln (I2C).
2. Filtern (Median-Filter).
3. Abstand auf dem Display anzeigen lassen (SPI).

Wenn die verschiedenen Funktionen sequenziell aufgerufen werden, dann muss das Program warten, bis N Messungen fertig sind und erst danach die Werte filtern und Anzeigen lassen. Da der I2C-Buss mit einer Frequenz von $40kHz$ arbeitet, während der Spartanmc Kern mit $60MHz$ arbeitet, bedeutet das, dass der Kern sehr lange nichts tut als auf die I2C Schnittstelle zu warten, bis sie fertig ist. Diese Zeit kann für andere Operationen ausgenutzt werden, z.B um mit dem Display zu kommunizieren oder die Messungen zu filtern. Hier kommt ein Interruptkontroller zum Einsatz. Wir verbinden ihn mit der I2C-Schnittstelle um am Ende jeder Übertragung einen Interrupt zu bekommen und die nächste I2C Funktionalität zu starten. Wir hätten den Interruptkontroller auch mit der SPI-Schnittstelle verbinden können, allerdings ist es weniger Sinvoll. Ein Grund

ist, dass die Kommunikation nur von dem SPI-Master zum SPI-Slave funktioniert. Ein weiterer Grund ist dass der SPI-Buss mit $10MHz$ betrieben wird und damit in der Größenordnung der Kernfrequenz arbeitet. Außerdem wird der Prozessorkern in dieser Zeit nichts anderes tun als warten, da in dieser Zeit auf einen Interrupt von der I2C-Schnittstelle gewartet wird. Man könnte behaupten, dass man den Filter anwenden kann um die Liste zu sortieren, allerdings wird der Wert auf der Anzeige aktualisiert in jedem Fall erst wenn die SPI-Peripherie zur Kommunikation zur Verfügung steht.

4.3.1 State Machine

Da die I2C-Schittstelle aus mehreren Schritten besteht, haben wir für das Interrupt-Service-Routine (ISR) eine Statemachine. Um in die Statemachine zu kommen, rufen wir das erste mal die I2C-Schreibefunktion in der Mainfunktion vor der While-Schleife. Diese verschiedenen Zustände werden im folgeneden beschrieben (Abbildung 4.1 veranschaulicht den Ablauf):

- Zustand $S1$: Das ist der Initialzustand. Das Program landet in diesem Zustand, nachdem ein Schreibevorgang beendet wurde. Dieser Schreibevorgang kann entweder eine Messung starten oder abfragen ob eine Messung fertig ist. Wenn eine Messung gestartet wurde und der Schreibevorgang dazu fertig ist, landen wir in dem ersten Fall. Da wird eine neue Übertragung gestartet, um abzufragen, ob eine Messung fertig ist. Die Variable ´is_write´ auf Null gesetzt, um beim nächsten Interrupt nicht da zu landen. Beim nächsten Interrupt wird dann geprüft, ob der Slave bei der ersten Abfrage ein Acknowledgement geschickt hat. Wenn der Sensor gerade den Abstand misst, dann werden die Abfragen vom Master ignoriert. Da erwarten wir kein Acknowledgement zu bekommen. Dieser Vorgang wird wiederholt, bis ein Acknowledgement ankommt. In diesem Fall landen wir in dem letzten Fall, um dort einen Schreibevorgang zu starten, um später die Messung vom Sensor zu lesen. In dem nächsten Interrupt landet das Program in Zustand $S2$.
- Zustand $S2$: In diesem startet das Program den Lesevorgang. Am Ende dieses Vorgangs stehen die Messungen zur Verfügung in dem Datenregister. Beim nächsten Interrupt landen wir in Zustand $S3$.
- Zustand $S3$: In diesem Zustand werden die abgelesenen Messungsbytes, die in dem Datenregister stehen, in einen Abstandswert umgerechnet und in der Liste der Messungen gespeichert. In diesem Zustand wird ein neuer Schreibevorgang gestartet, um eine neue Messung zu beginnen. Der nächste Zustand ist $S1$.

Die Implementierung des ISR mit der Statemachine sehen wir in Codeauschnitt unten. An dieser Stelle wollen wir in mehr Details die ifelse Abgfrage in Zustand $S1$ diskutieren. Laut der Dokumentation des Ultraschallsensors, kann man prüfen ob der Sensor gerade misst, indem man versucht, das erste Register vom Sensor zu lesen, dort steht die Firmwareversion. Wenn in dieser Zeit eine Messung durchgeführt wird, dann wird in das Datenregister vom Master die Zahl 255 geschrieben. Wenn aber der Sensor nicht misst, dann wird die Version der Firmware geschickt. Das bedeutet, dass man während einer Messung abfragen kann ob der Sensor fertig ist, indem man die Firmwareversion abfragt. Wir haben versucht diese Funktionalität zu implementieren

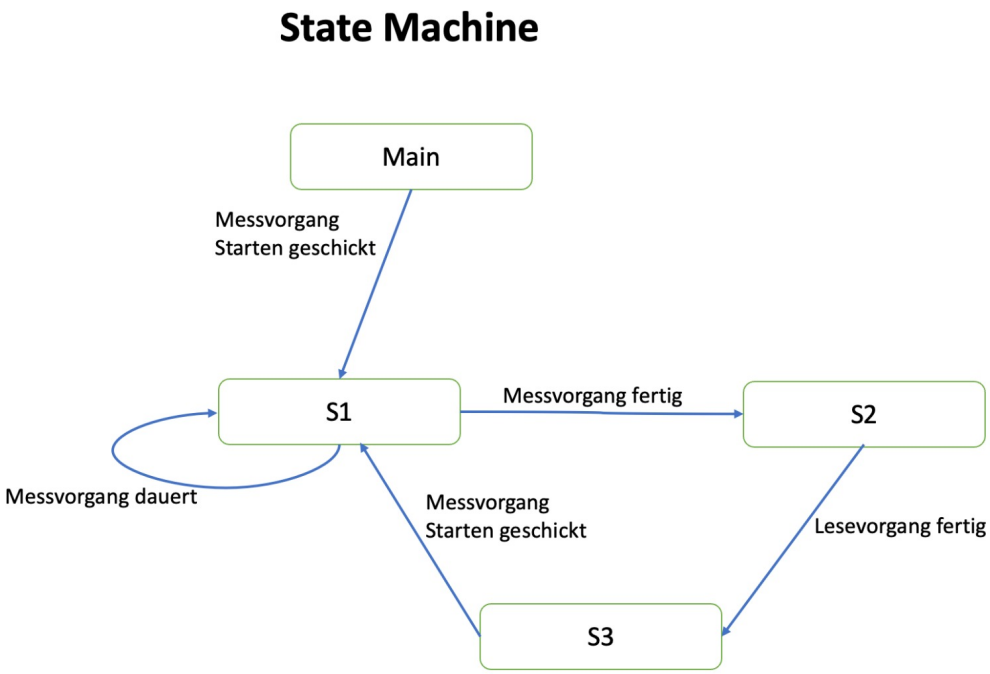


Abbildung 4.1: State Machine

aber wir haben festgestellt, dass bei der Abfrage die Zahl 255 in das Datenregister nie geschrieben wurde (der auskommentierte Code in Zustand $S1$ gehört dazu, is measuring muss dann auch entsprechend angepasst werden). Auch bei der unserer jetztigen Implementierung haben wir festgestellt, dass immer ein Acknowledgement zurückgeliefert wird. Dieses Verhalten könnte bedeuten, dass die Messungen sehr schnell beendet werden. Grund hierfür können die kurzen Abstände sein, die wir gemessen haben. Der maximale Abstand, den wir gemessen haben war gut $2m$ (wegen der Dimensionierung von dem Laborraum). Der Sensor ist in der Lage, bis $6m$ zu messen. Es kann sein, dass bei anderen Bedingungen und bei Messungen von $6m$ der Sensor länger messen muss und da auch die Zahl 255 zurückliefern würde. Um grob feststellen zu können, wie lange eine Messung dauert, haben wir einen Logikanalyzer verwendet. Dort konnten wir sehen, dass die Pollingabfrage sofort ein Acknowledgement liefert. Danach wird die Messung Erfolgreich abgelesen. In einer anderen Version von Zustand $S1$, haben wir sofort die Lesesequenz gestartet (also Registernummer schreiben und dann 2 Bytes lesen). Wir haben erwartet, da Probleme zu bekommen, weil wir nicht auf das Ende der Messung warten, allerdings auf der Anzeige konnten wir sinvolle Messungen feststellen. Das könnte auch heißen, dass der Sensor bei kleinen Abständen die Messung sehr schnell erledigt.

```
1
2 void is_measuring(i2c_master_regs_t* i2c_master)
3 {
4     // interrupt ausloesen
5     i2c_master->cmd  = 0; //clear all command register bits == 0
6     i2c_master->data[0] = I2C_SLAVE_ADDRESS; //for write
7     i2c_master->cmd = I2C_CMD_STA | (1 << 3) | I2C_CMD_STO;
```

```
}

// interrupt service routine
ISR(0)()
{
    switch(state_i2c)
    {
        case S1: // wait measurmement
            if(is_write)
            {
                is_measuring(&I2C_MASTER);
                is_write = 0;

            }
            else if(I2C_MASTER.stat & I2C_STA_NO_ACK)
            {
                is_measuring(&I2C_MASTER);

            }
            else
            {
                i2c_peri_set_sense_register(&I2C_MASTER, 0x02);
                state_i2c = S2;
            }

            /* The code in this comment is for checking firmware version number
            if(is_write)
            {
                is_measuring(&I2C_MASTER);
                is_write = 0;

            }
            else if(I2C_MASTER.data[1] == 255)
            {
                is_measuring(&I2C_MASTER);

            }
            else
            {
                i2c_peri_set_sense_register(&I2C_MASTER, 0x02);
                state_i2c = S2;
            }*/
            break;
        case S2:// set read register
            i2c_peri_read_start(&I2C_MASTER, I2C_SLAVE_ADDRESS);
            state_i2c = S3;
            break;
        case S3: // calculate the distance and start a new measurement
            dis_array[meas_n] = i2c_peri_read_distance(&I2C_MASTER);
            meas_n++;
            if(meas_n >= MEAS_N) // get new distance when 10 new measurements are
   collected
            {
```

```
                meas_n = 0;
            }
            i2c_peri_write(&I2C_MASTER);
            is_write = 1;
            new_val = 1;
            state_i2c = S1;
            break;
        default:
            printf("shouldnt occur");
            break;
    } // end of switch(state)
}
```

ISR

5 Zusammenfassung

In diesem Versuch haben wir Treiber zu Kommunikationsprotokolen zwischen Hardware Komponenten implementiert. Diese sind der SPI-Treiber und I2C-Treiber.

Bei dem SPI Treiber zur Kommunikation mit dem Display haben wir nur die Seite vom Master implementiert und die SPI-Schnittstelle wurde in 3 Wire Modus betrieben. Wir konnten erfolgreich verschiedene Informationen auf dem Display anzeigen lassen und das Display steuern. Die SPI-Kommunikation ist in unserem Fall Unidirektional. Das bedeutet, dass die Information nur von dem SPI-Master zu dem SPI-Slave fließt.

Bei der Implementierung von dem I2C haben wir auch nur die Master-Seite Implementiert. In dieser Schnittstelle ist die Kommunikation Bidirektional. Dafür wurden Lese-und Schreibefunktionen Implementiert.

Nach der Implementierung beider Kommunikationsprotokolle, wurden beide in einem System integriert. Ziel ist es die Abstandsmessungen auf dem Display anzeigen zu lassen. Dabei dient die I2C-Schnittstelle zur Kommunikation mit dem Ultraschallsensor und die SPI-Schnittstelle zur Kommunikation mit dem Display.

Bei der Integration beider Treibern haben wir die I2C-Schnittstelle um eine ISR erweitert, um auf die Pollingabfragen verzichten zu können und diese Zeit für andere Berechnungen auszunutzen. Dafür haben wir auch eine Statemachine implementiert. Am Ende der Implementierung konnten wir feststellen, dass die Messungen schneller auf dem Display aktualisiert werden, im Vergleich zu der Implementierung mit Pollingabfragen.

Bei der Abfrage vom Sensor, ob die Messung schon fertig ist, konnten wir nicht feststellen, ob der Sensor tatsächlich irgendwann die Zahl 255 zurückliefert während einer Messung. Stattdessen haben wir immer die Firmwareversion bekommen (die Nummer 6). Danach haben wir uns entschieden zu prüfen, ob wir ein Acknowledgement bei der Abfrage bekommen, da der Sensor während einer Messung nicht reagiert und somit kein Acknowledgement senden kann. Hier haben wir bei Abfrage Acknowledgement bekommen. Dieses Verhalten könnte bedeuten, dass die von uns durchgeführten Messungen schnell erledigt werden. In dem Labor haben wir es geschafft, Distanzen von bis zu $2m$ messen können. Man kann in einem anderen Raum auch testen, ob Messungen von größeren Distanzen länger dauern.

Da die Messungen vom Sensor auch Außreiser enthalten, haben wir einen Medianfilter implementiert. Der von uns implementierte Medianfilter konnte mit den Außreisern gut filtern. Um den Medien aus den gegebenen Messungen berechnen zu können, haben wir einen Sortieralgorithmus

implementiert. Dieser sortiert die Liste der Messungen bevor der Median berechnet wird. In diesem Versuch haben die Quicksort und Bubblesort Algorithmen implementiert und verglichen. Dabei konnten wir keinen unterschied beim Aktualisieren der Anzeige feststellen. Das kann damit begründet werden, dass die Länge der Liste klein ist oder bereits fast sortiert ist. In Zukunftigen arbeiten kann man testen, wie die Algorithmen mit längeren Listen umgehen.

Literatur

[1] *Bubblesort*. URL: https://en.wikipedia.org/wiki/Bubble_sort.
[2] *Quicksort*. URL: https://de.wikipedia.org/wiki/Quicksort.